
Datenstrukturen, Algorithmen und Programmierung II

Prof. Dr. Petra Mutzel
Markus Chimani
Carsten Gutwenger
Karsten Klein

Skript zur gleichnamigen Vorlesung von
Prof. Dr. Petra Mutzel

im Sommersemester 2006

Kapitel 8

Geometrische Algorithmen

In der Praxis kommt es immer wieder vor, dass geometrische Daten gegeben sind, d.h. Daten, die gewisse Koordinaten im k -dimensionalen Raum besitzen. Beispiele dafür sind Städte auf einer Landkarte, Komponenten im Chipdesign, u.v.m. In der *Algorithmische Geometrie* (engl. *Computational Geometry*) nutzen wir die geometrischen Eigenschaften der Daten aus, um effiziente Algorithmen zu erhalten. Die Anwendungsgebiete reichen von geographischen Informationssystemen, Chip-Layout und Sensornetzwerken, über Bildverarbeitung, CAD und Robotik bis zu Computergraphik und Computerspielen.

Wir werden uns in diesem Kapitel auf zwei grundlegende Bereiche beschränken: Zunächst werden wir anhand eines Beispielproblems das Prinzip der Sweepline-Algorithmen erläutern, die einen wesentlichen Baustein vieler fortgeschrittenerer Algorithmen bilden. Danach lernen wir grundlegende Datenstrukturen zur Speicherung von zwei-, drei- oder auch mehrdimensionalen Daten kennen, in denen sich sogenannte Bereichssuchen effizient durchführen lassen.

8.1 Sweepline Algorithmen

Viele zweidimensionale Probleme lassen sich durch den Ansatz der als *Sweepline Prinzip* bekannt ist, effizient lösen. Sie basiert darauf, dass eine Sweepline (manchmal auch *Scanline* genannt) über die Fläche auf der die Datenpunkte liegen parallelverschoben wird. Anstatt das ursprüngliche Problem auf der zweidimensionalen Fläche zu lösen, betrachten wir immer nur Ereignisse die auf dieser sich bewegenden Sweepline auftreten. Wir verwandeln also ein zweidimensionales Problem in eine *Folge* von eindimensionalen Problemen, die sich dann jeweils recht einfach lösen lassen.

Nehmen wir in Folge an, dass wir eine vertikale Sweepline L von links nach rechts über unsere Fläche führen. Unabhängig von der genauen Problemstellung teilt sie unsere Objekte zu jedem Zeitpunkt in drei verschiedene Klassen ein:

- *Abgeschlossene Objekte*: Objekte, die vollständig links von L liegen, und dadurch schon vollständig von der Sweepline behandelt wurden.
- *Aktive Objekte*: Objekte, die gegenwärtig von L geschnitten werden, d.h. gerade für die Berechnung eines eindimensionalen Teilproblems herangezogen werden.
- *Wartende Objekte*: Objekte, die vollständig rechts von L liegen, also bisher noch nicht von der Sweepline behandelt wurden.

Die Menge der aktiven Objekte – ggf. ergänzt durch zusätzliche Daten – wird auch als *Sweepline-Status* bezeichnet.

Algorithmisch ist es natürlich recht sinnfrei, eine Linie kontinuierlich über eine Ebene streifen lassen zu wollen, und das ist für das Sweepline-Prinzip natürlich auch nicht notwendig. Wir interessieren uns immer für diejenigen Positionen der Sweepline, an denen sich tatsächlich etwas ändert. Diese Stellen werden *Ereignispunkte* genannt, und statt die Linie nun kontinuierlich zu bewegen, müssen wir sie nur von links nach rechts von einem Ereignispunkt zum nächsten springen zu lassen. Ein wesentlicher Teil der Entwicklung und Analyse von Sweeplinealgorithmen basiert also darauf, die Zahl der notwendigen Ereignispunkte möglichst klein zu halten.

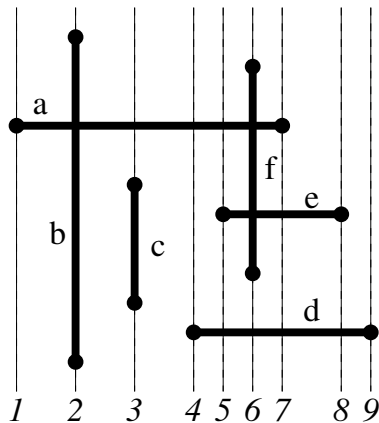
Die offensichtlichsten Ereignispunkte sind diejenigen Stellen, an denen ein Objekt seine Klassifizierung ändert, also wenn ein wartendes Objekt aktiv, oder ein aktives Objekt abgeschlossen wird. Offensichtlich lassen sich diese Ereignispunkte recht einfach statisch berechnen, bevor wir anfangen die Sweepline tatsächlich zu bewegen. Problemabhängig kann es auch noch weitere *dynamische* Ereignispunkte geben, die wir erst während der Sweeplinebewegung zu unserer Ereignismenge hinzufügen können.

Um dieses ganz allgemein gehaltene Prinzip zu verdeutlichen, wenden wir uns in Folge beispielhaft dem Schnittproblem für iso-orientierte Liniensegmente zu.

8.1.1 Beispiel: Schnittproblem für iso-orientierte Liniensegmente

Schnittproblem für iso-orientierte Liniensegmente	
<i>Gegeben:</i>	eine Menge $S = \{s_1, \dots, s_n\}$ von vertikalen und horizontalen Liniensegmenten in zweidimensionalen Raum
<i>Gesucht:</i>	alle Paare sich schneidender Segmente

Ein Beispiel für eine solche Problemstellung ist in Abbildung 8.1 gezeigt.



<i>E.Pkt.</i>	<i>Typ</i>	<i>s =</i>	<i>A =</i>	<i>Ausgabe</i>
1	E1	<i>a</i>	$\langle a \rangle$	
2	E3	<i>b</i>	\vdots	(b, a)
3	E3	<i>c</i>	\vdots	\emptyset
4	E1	<i>d</i>	$\langle d, a \rangle$	
5	E1	<i>e</i>	$\langle d, e, a \rangle$	
6	E3	<i>f</i>	\vdots	$(f, e), (f, a)$
7	E2	<i>a</i>	$\langle d, e \rangle$	
8	E2	<i>e</i>	$\langle d \rangle$	
9	E2	<i>d</i>	\emptyset	

Abbildung 8.1: Eine Probleminstanz für den Schnitt von iso-orientierten Liniensegmenten. Die Tabelle rechts zeigt die entsprechenden Werte im Ablauf des Sweepline-Algorithmus.

Naive Lösung. Die wohl erste Idee dieses Problem zu lösen basiert darauf, dass wir einfach für jedes Paar von Segmenten testen, ob sie sich schneiden. Die Laufzeitanalyse für dieses Vorgehen sollte zum jetzigen Zeitpunkt nicht mehr schwer fallen: Das erste Segment muss mit den anderen $n - 1$ Segmenten verglichen werden, das zweite mit $n - 2$, usw. Dies führt natürlich zu einer Laufzeitabschätzung von $\Theta(n^2)$.

Worst-Case. Es lassen sich tatsächlich Beispiele finden, in denen n iso-orientierte Segmente $\Theta(n^2)$ viele Kreuzungen besitzen. Da wir alle Kreuzungen ausgeben wollen, können wir uns in diesem Fall keine bessere Laufzeit erwarten. Ist also die naive Idee vielleicht auch schon das Optimum das erreichbar ist? Der Nachteil des naiven Verfahrens liegt daran, dass es – unabhängig von der tatsächlichen Anzahl der Schnittpunkte – immer ein quadratisches Laufzeitverhalten zeigt. Wünschenswerter wäre ein *output sensitiver* Algorithmus, der nur dann quadratische Laufzeit benötigt, wenn die Schnittanzahl tatsächlich auch quadratisch ist, und ansonsten wesentlich schneller läuft. Mit dem Sweepline-Prinzip haben wir ein Werkzeug in der Hand, das uns dies ermöglichen wird.

Da es uns nur darum geht, das Sweepline-Prinzip zu verdeutlichen, werden wir uns die Aufgabenstellung dahingehend vereinfachen, dass wir annehmen, dass alle Anfangs- und Endpunkte der horizontalen Segmente, und alle vertikalen Segmente paarweise verschiedene x -Koordinaten haben.

Anmerkung 8.1. Im Allgemeinen können Spezialfälle wie zusammenfallende Koordinaten oft viel kniffliger zu behandeln sein, als das eigentliche Aufstellen des zugrundeliegenden Sweepline-Algorithmus. Auch wenn im vorliegenden Fall diese Spezialfälle nicht allzu kompliziert zu behandeln wären, sehen wir aus Gründen der Klarheit doch davon ab.

In Folge entwickeln wir den einen Sweepline-basierten Algorithmus, der in Listing 8.1 dargestellt ist. Es ist klar, dass Schnitte nur zwischen jeweils einem horizontalen und einem vertikalen Segment auftreten können. Ausgehend vom allgemeinen Sweepline-Prinzip, können wir folgende Beobachtung machen:

Beobachtung 8.1. *Trifft die Sweepline L auf ein vertikales Liniensegment s , so kann sich s nur mit den derzeit aktiven (horizontalen) Segmenten schneiden.*

Die Überprüfung mit welchen der aktiven Segmente sich s nun tatsächlich schneidet ist die oben erwähnte Reduktion auf ein eindimensionales Problem: wir müssen einfach nur überprüfen ob die y -Koordinate eines horizontalen Segments zwischen den y -Koordinaten der Endpunkte von s liegt.

Für jede y -Koordinate der vertikalen Segmente müssen wir also wissen welche horizontalen Kantensegmente gerade aktiv sind. Dies führt zu folgenden drei Typen von Ereignispunkten:

- (E1) Für alle horizontalen Segmente: x -Koordinate des linken Endpunkts. An dieser Stelle wird das Segment von einem wartenden Objekt zu einem aktiven.
- (E2) Für alle horizontalen Segmente: x -Koordinate des rechten Endpunkts. An dieser Stelle wird das Segment von einem aktiven Objekt zu einem abgeschlossenen.
- (E3) Für alle vertikalen Segmente: x -Koordinate. An dieser Stelle werden die Schnitte mit den aktiven horizontalen Segmenten berechnet.

Diese Ereignispunkte lassen sich durch ein einfaches Ablaufen der Eingabedaten ermitteln; es gibt $\Theta(n)$ viele davon. Da die Sweepline sich von links nach rechts bewegen soll, müssen wir diese Punkte noch sortieren. Wir wissen, dass dies in $O(n \log n)$ möglich ist.

Schließlich müssen wir uns nur noch überlegen, wie wir die Menge der aktiven horizontalen Segmente (A) verwalten wollen. Wir müssen folgende Operationen effizient unterstützen:

- Einfügen eines Elements (bei Ereignispunkten des Typs E1)
- Entfernen eines Elements (bei Ereignispunkten des Typs E2)
- Alle Elemente bestimmen, die in einen gegebenen Bereich $[y_{\min}, y_{\max}]$ fallen (bei Ereignispunkten des Typs E3)

Die dritte Operation ist notwendig, um nicht für alle aktiven Segmente tatsächlich den Schnitttest durchführen zu müssen: Wenn A eine nach y -Koordinate geordnete Menge ist, genügt es das kleinste Segment zu finden, das in diesen Bereich fällt. Danach können wir in aufsteigender Reihenfolge solange alle weiteren Segmente hinzunehmen, bis wir auf ein Segment treffen, das nicht mehr in den gegebenen Bereich fällt.

Eingabe: Menge S von horizontalen und vertikalen Liniensegmenten, mit paarweise unterschiedlichen x -Koordinaten	
Ausgabe: Alle Paare von sich schneidenden Liniensegmenten	
1:	Bestimme die Menge der Ereignispunkte Q aus S [$O(n)$]
2:	Sortiere Q nach aufsteigender x -Koordinate [$O(n \log n)$]
3:	$A := \emptyset$
4:	for all Ereignispunkte $p \in Q$ do ▷ von links nach rechts...
5:	$s :=$ Liniensegment das zu p gehört
6:	if s ist horizontal then
7:	if p ist linker Endpunkt von s then
8:	Füge s in A ein;
9:	else ▷ p ist rechter Endpunkt von s
10:	Entferne s aus A ;
11:	end if
12:	else ▷ p ist x-Koordinate eines vertikalen Segments s
13:	$y_{\min}, y_{\max} :=$ y -Koordinaten der Endpunkte von s
14:	Bestimme alle Segmente $t \in A$, deren y -Koordinate im Bereich $[y_{\min}, y_{\max}]$ liegen und gib (s, t) als Paar sich schneidender Segmente aus
15:	end if
16:	end for

Listing 8.1: Sweepline-Algorithmus zur Lösung des Schnittproblems von iso-orientierten Liniensegmenten

Nachdem an jedem Ereignispunkt genau eine dieser Operationen anfällt, und jeder Ereignispunkttyp $O(n)$ mal auftritt, suchen wir eine Datenstruktur, bei denen die langsamste dieser Operationen möglichst schnell ist.

Dafür bieten sich balancierte Suchbäume, z.B. AVL-Bäume, an: Einfügen und Entfernen ist in jeweils $O(\log n)$ möglich. Die Bereichsabfrage ist in $O(\log n + r)$ möglich, wenn r die Anzahl der zu findenden Elemente darstellt: wir erzielen diese Laufzeit beispielsweise durch eine Suche des kleinsten Elements, das in den Suchbereich fällt ($O(\log n)$) und laufen den Baum danach in Inorder-Reihenfolge ab, bis wir auf ein Element mit zu großer y -Koordinate treffen ($O(r)$).

Laufzeit des Sweepline-Algorithmus. Wir besuchen jeweils $O(n)$ Ereignispunkte des Typs E1, E2 und E3. In den beiden ersten Fällen benutzten wir eine Operation mit logarithmischer Laufzeit $O(\log n)$, im dritten Fall benötigen wir die etwas sperrigere Laufzeit $O(\log n + r)$. Nun ist aber klar, dass unser Algorithmus jeden Schnittpunkt genau einmal findet, d.h. die Summe über all die verschiedenen r ist insgesamt R , die Anzahl der Schnitte. Dadurch können wir die r aus dem Produkt mit $O(n)$ herausziehen, und erhalten: $O(n \log n + R)$.

Wir sehen, dass die resultierende Laufzeit nun nicht mehr nur von der Eingabegröße, sondern auch von der Ausgabegröße – also der Anzahl der vorhandenen Schnittpunkte – abhängt. Da wir wissen, dass $R = O(n^2)$, sehen wir, dass unser Algorithmus im Worst-Case Fall tatsächlich eine quadratische Laufzeit besitzt – er ist jedoch nicht schlechter als der naive Algorithmus. Wächst die Anzahl der Schnittpunkte jedoch schwächer, z.B. linear, dann benötigt der Sweepline-Algorithmus nur $O(n \log n)$ Zeit. Der Platzbedarf des Algorithmus (exklusive der Ausgabe) ist in jedem Fall linear.

Anmerkung 8.2. Man kann formal zeigen, dass es keinen Algorithmus geben kann, der eine bessere obere Schranke für das Schnittproblem iso-orientierter Liniensegmente garantiert.

8.1.2 Erweiterungen

Auf ähnliche Weise lässt sich auch das analoge Problem für allgemeine (also nicht-iso-orientierte) Segmente lösen, allerdings ist es hier aufwendiger auch im Worst-Case-Fall die Laufzeit des naiven Verfahrens zu garantieren. Ein anderes in der Literatur sehr beliebtes Problem ist ebenfalls eine Erweiterung der Schnittberechnung von iso-orientierten Segmenten: Schnitte bzw. Inklusionen von (achsenparallelen) Rechtecken zu berechnen ist ein Problem das in vielen GUI Anwendungen auftritt. In diesem Zusammenhang sei auf die Literaturhinweise am Ende dieses Kapitels verwiesen.

Das Sweepline-Prinzip kann auf analoge Weise auch für höherdimensionale Räume angewendet werden, so können wir zum Beispiel eine *Sweep-Plane* durch den dreidimensionalen Raum gleiten lassen. D.h. man reduziert das dreidimensionale Problem auf eine Folge von zweidimensionalen Problemen; das besonders interessante dabei ist, dass man diese zweidimensionalen Probleme ihrerseits dann oft wieder mit einem Sweepline-Algorithmus behandeln kann.

8.2 Datenstrukturen zur Suche

Dieser Abschnitt wurde im Sommersemester 2006 nur oberflächlich behandelt und ist daher nicht klausurrelevant!

Als zweiten Teil unseres Ausflugs in die Welt der geometrischen Algorithmen wollen wir uns nun mit der *mehrdimensionale Bereichssuche* beschäftigen. Wir stellen grundlegende Ideen und Datenstrukturen vor, die es ermöglichen, in zwei-, drei- oder noch höherdimensionalen Daten effizient zu suchen.

Die genaue Problemstellung, die wir in weiterer Folge untersuchen wollen ist die folgende:

Mehrdimensionale Bereichssuche	
<i>Gegeben:</i>	Punktmenge $P = \{p_1, p_2, \dots, p_n\}$ im k -dimensionalen Raum und ein k -dimensionaler (achsenparalleler) Bereich D
<i>Gesucht:</i>	alle Punkte die in D liegen

Für $k = 1$ bedeutet diese Fragestellung z.B. alle Zahlen aus einer gegebenen Zahlenfolge zu finden die größer 5, aber kleiner 12 sind. Der Fall $k = 2$ entspricht z.B. der Fragestellung, alle Großstädte aufzuzählen, deren Breitengrad um maximal 15, und deren Längengrad um maximal 10 von Dortmund abweichen. Sehr viel höhere Dimensionen treten oft bei Datenbankabfragen auf: Wenn wir eine Liste aller Studeten Dortmunds die zwischen 22 und 26 Jahre alt sind, in Essen wohnen, im Nebenfach Anglistik studieren, und eine Telefonnummer haben die mit einer 4 beginnt, so ist dies auch nichts anderes als eine multidimensionale Bereichsabfrage.

Achsenparallele Bereichsabfragen über Punktmengen kommen also schon in ihrer puren Form recht häufig vor, doch der wahre Wert der Untersuchung ergibt sich daraus, dass die hier vorgestellten Techniken sich auch einfach erweitern lassen, um z.B. in nicht-achsenparallelen Bereichen allgemeinere Objekte als nur Punkte, zu finden, bzw. nach ihrer Lage im Raum zu klassifizieren.

Naive Methode. Die sicherlich einfachste und naivste Methode dieses Problem zu lösen ist es, jeden Punkt darauf zu testen, ob seine Koordinaten innerhalb von D liegen. Dieser Test dauert natürlich jeweils $O(k)$, da wir jede Koordinate testen müssen, um sicherzugehen dass ein Punkt in D liegt. Diesen Test für jeden Punkt durchzuführen bedeutet in Summe eine Laufzeit $O(kn)$.

Mehrere Abfragen. Für den Fall $k = 1$ ist das naive Verfahren recht analog zur lineare Suche, und so fällt uns sofort auf, dass wir mit der binären Suche schon eine weitaus effizientere Methode kennengelernt haben (vergl. auch die Datenstruktur für aktive Objekte beim vorgestellten Sweepline-Algorithmus). Allerdings erinnern wir uns, dass wir zum Anwenden dieser Methode das Zahlenfeld zunächst sortieren mussten, also die gegebenen Daten aufbereiten mussten, um die effiziente Suche zu ermöglichen.

Wenn wir nur eine einzige Abfrage über unsere Datenpunkte machen, zahlt sich dieses *Pre-processing* nicht aus, denn wir benötigen $O(n \log n) + O(\log n) + O(r) = O(n \log n)$ Schritte für die Sortierung, das Auffinden des kleinsten in den Bereich fallenden Elements, und das Ablaufen aller weiterer in den Bereich fallender Elemente (r). Dies ist schlechter als die naive Methode die jedes Element genau einmal betrachtet ($O(n)$).

Allerdings ist auch klar, dass sich der Aufwand des Sortierens der Daten lohnt, sobald „genug“ verschiedene Abfragen (d.h. verschiedene Bereiche D) auf der selben Punktmenge ausgeführt werden. Nehmen wir der Einfachheit halber an, dass r immer „recht

klein" ist, also konstante Grösse hat. Wenn wir m Abfragen durchführen, benötigt das naive Verfahren insgesamt $\Theta(mn)$ Schritte, während die binäre Suche eine Laufzeit von $O(n \log n) + mO(\log n) = O(\max(m, n) \log n)$ ermöglicht. Das Preprocessing zahlt sich also aus, sobald mehr als $\log n$ viele Abfragen durchgeführt werden. In weiterer Folge gehen wir von der durchaus realistischen Annahme aus, dass wir – dieser Betrachtung folgend – immer „genug“ Abfragen erwarten können, um ein Preprocessing zu rechtfertigen.

Ist r , die Anzahl der gefundene Punkte, bei jeder Anfrage ein konstanter Anteil aller Punkte, so ist die naive Methode asymptotisch optimal. Liefert aber jede Suchanfrage nur eine kleine konstante Anzahl von Punkten als Ergebnis, so ist das Verfahren wie eben gezeigt sehr ineffizient. Wir wollen wieder einen *output sensitiven* Algorithmus entwickeln, bei dem die Laufzeit von der Anzahl der im Suchbereich liegenden Punkte und somit von der Größe der Ausgabe abhängig ist.

Ziel ist es nun, auch bei höheren Dimensionen möglichst analoge Einsparungen wie im Fall $k = 1$ zu erzielen. Wir werden zunächst immer den Fall $k = 2$ besprechen, und danach darlegen, wie sich die Ideen für $k > 2$ verallgemeinern lassen.

8.2.1 Gitter und Quadrees

Gittermethode. Die einfachste Idee die Laufzeit zu beschleunigen ist die sogenannte *Gittermethode*. Hierbei wird die Fläche auf der die Punkte liegen durch ein regelmässiges Gitter in mehrere Bereiche aufgeteilt (vergl. Abb. 8.2). Die Punkte werden dann zu den entsprechenden Bereichen gespeichert. Erfolgt nun eine Abfrage über den Bereich D , so müssen zunächst nur die Gitterbereiche B an denen D teilweise beteiligt ist, bestimmt werden. Da das Gitter regelmässig ist, kann eine derartige Berechnung sehr einfach durchgeführt werden. Danach müssen nur mehr die Punkte die in Bereichen von B liegen, geprüft werden.

Es ist klar, dass dieses Verfahren im Allgemeinen zu keiner Verbesserung der asymptotischen Laufzeit führen kann, da – bei konstanter Anzahl der Gitterlinien – in jedem Gitterbereich $O(n)$ Punkte liegen. Auch kann es passieren, dass sämtliche Punkte (bis auf einen), in einem einzigen Gitterbereich landen.

In vielen Fällen kann man aber davon ausgehen, dass die Punkte ungefähr gleichverteilt sind, so dass sich die Gittermethode dazu eignet, die *Konstanten*, die in der O -Notation ja unterdrückt werden, zu senken. D.h. die Gittermethode *kann*, wenn sie richtig angewendet wird, in der Realität einen Algorithmus um nennenswerte konstante Faktoren beschleunigen.

Höhere Dimensionen. Die Gittermethode bietet sich natürlich auch für grössere Dimensionen an. Anstatt eine zweidimensionale Fläche in eine Menge von kleinen Quadraten zu zerlegen, kann man entsprechend auch einen dreidimensionalen Raum in eine Menge von kleinen Würfelchen zerschneiden, usw.

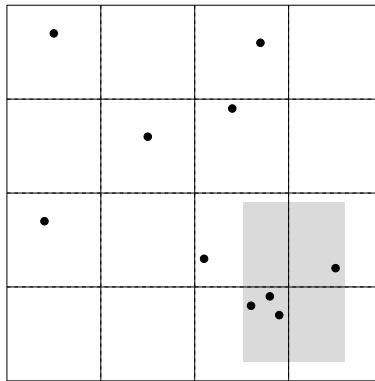


Abbildung 8.2: Gittermethode.

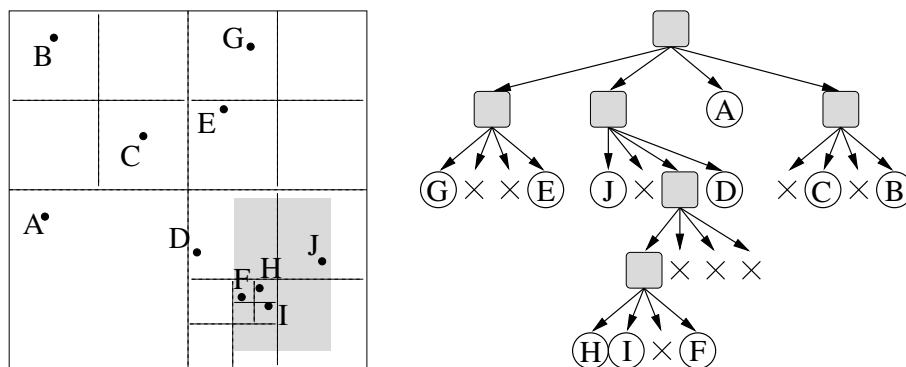


Abbildung 8.3: Quadtree.

Anmerkung 8.3. Die Gittermethode findet unter anderem sogar Anwendung in der Archäologie! Bei Ausgrabungen wird ein Gebiet z.B. durch farbige Schnüre in Parzellen aufgeteilt, so dass man die ausgegrabenen Gegenstände nachher einfach der Parzelle zuordnen kann, in der man sie gefunden hat.

Quadtree. Eine Schwierigkeit bei der Gittermethode ist es, die beste *Auflösung* des Gitters zu bestimmen: Haben wir zu wenige/zu große Gitterregionen, sind die Geschwindigkeitsvorteile nicht besonders groß. Haben wir jedoch zu viele/zu kleine Gitterregionen, so wird der Speicheraufwand enorm. Vorallem müssen wir bei zu feinem Gitter damit rechnen, dass sehr viele Regionen leer sind, und bei der Bereichssuche einen beträchtlichen Overhead verursachen.

Die einfachste Idee dieses Problem zu umgehen ist der sogenannte *Quadtree*: Wir nehmen an, dass wir mehr als einen Punkt gegeben haben, und gehen von unserer Grundfläche G aus. Zerteilen wir G nun in der Mitte sowohl horizontal, als auch vertikal, so erhalten wir vier

Teilregionen G_1, \dots, G_4 , d.h. wir legen defakto also ein extrem grobes Gitter – nämlich eines, dass uns genau vier Gitterregionen definiert – über die Grundfläche.

Nun betrachten wir die vier neuen Regionen einzeln und zählen die Punkte die in ihnen liegen. Für jede Region G_i in der mehr als ein Punkt liegt, führen wir wieder eine Zerteilung in vier Teilregionen $G_{i,1}, \dots, G_{i,4}$ durch, usw. (vergl. Abb. 8.3).

So definiert sich ein eine Baumstruktur in der jeder innere Knoten genau vier Kinder hat. Eine Ebene nach unten zu steigen bedeutet, ein doppelt so feines Gitter im entsprechenden Teilgebiet zu definieren. In Gebieten, in denen die Punkte einen grossen Abstand zueinander haben, benötigen wir nur ein recht grobes Gitter. In Gebieten in denen die Punkte dicht liegen, steht uns ein feineres Gitter zur Verfügung.

Auf dem Weg zu einer effizienten Datenstruktur zur Bereichsuche kann leider auch diese Methode noch keine Verbesserung in der Laufzeit erbringen. Bei sehr ungleichmässig verteilten Punkten kann die Datenstruktur sogar so entarten, dass ihre Größe unbeschränkt ist! Wenn wir z.B. alle bis auf einen Punkt sehr dicht nebeneinander legen, so entsteht ein Baum, für den man im theoretischen Modell das Gebiet quasi unendlich oft zerteilen muss, bis es die Punktmenge tatsächlich aufzuspalten vermag. Auf jeden Fall ist seine Tiefe (und damit auch seine Gesamtgröße) durch keine Funktion in n beschränkt. Es lässt sich allerdings zeigen, dass die Tiefe maximal $\frac{3}{2} + \log \frac{s}{c}$ sein kann, wobei $\frac{s}{c}$ den Quotienten zwischen der Seitenlänge der Grundfläche und der kleinsten Distanz zwischen zwei Punkten angibt.

Anmerkung 8.4. Durch ein paar Kniffe lassen die Probleme der zu starken Entartung umgehen; dies geschieht allerdings auf Kosten der Einfachheit der Datenstruktur. Es sei nur erwähnt, dass auch dadurch die O -Notation der Bereichssuche nicht gedrückt werden kann.

Anmerkung 8.5. In vielen Anwendungen ist es effizienter, den Quadtree nicht solange aufzubauen, bis in jeder Region nur mehr maximal ein Punkt liegt. Meist beendet man die Aufteilung, sobald maximal m Punkte in einer Region enthalten sind, wobei m eine durch Experimente für gut befundene Konstante darstellt. Oft baut man den Baum auch nur bis zu einer konstant gewählten Maximaltiefe auf.

Höhere Dimensionen. Analog zur Erweiterung der Gittermethode, können wir auch Quadrees dadurch erweitern, dass wir im dreidimensionalen Fall einen grossen „Grundwürfel“ immer wieder in acht kleinere Teilwürfel zerlegen. Da jeder innere Knoten nun immer acht Kinder hat, wird der so entstehende Baum *Octree* genannt. Wir erkennen allerdings, dass wir bei k Dimensionen 2^k Kinder verwalten müssen, die Kinderanzahl also exponentiell wächst. Daher wird die Datenstruktur in der Regel nicht für mehr als drei Dimensionen eingesetzt.

Quadrees in der Praxis. Auch wenn sich Quadrees nicht dazu eignen, die formale asymptotische Laufzeit der Bereichsabfrage zu senken, so können sie dennoch die an der Laufzeit beteiligten konstanten Koeffizienten senken. Darüber hinaus haben sie besonders hilfreiche

Eigenschaften, wie z.B. dass die Auflösungen des Gitters immer Zweierpotenzen entspricht und die Grundstruktur sehr einfach ist. Dies erlaubt einfache und vorallem in der Praxis sehr schnelle Algorithmen. Vorallem in der Computergraphik und bei Computerspielen bilden Quad- und Octrees einen fixen Bestandteil des Erfolgsgeheimnisses.

8.2.2 k D-Baum und BSP-Baum

Vergleichsoperationen von Koordinaten. Auch wenn die Punkte unserer Punktmenge dank der Mengendefinition paarweise unterschiedlich sind, so können doch zwei verschiedene Punkte $p_i = (x_i, y_i)$ und $p_j = (x_j, y_j)$ z.B. die gleiche x -Koordinate haben. Oft – so auch im Fall der in Folge vorgestellten Datenstruktur – ist es in der algorithmischen Geometrie angenehmer wenn man auch voraussetzen kann, dass alle x - und alle y -Koordinaten jeweils untereinander unterschiedlich sind. Dies kann durch eine einfache *erweiterte Ordnung* erreicht werden, in dem man die Vergleichsoperationen der x - bzw. y -Koordinaten erweitert: sind zwei Punkte bezüglich der zu vergleichenden Koordinate gleich, so entscheiden wir auf Basis der anderen Koordiante. Formal bedeutet das:

$$p_i <_x p_j \Leftrightarrow x_i < x_j \vee (x_i = x_j \wedge y_i < y_j)$$

$$p_i <_y p_j \Leftrightarrow y_i < y_j \vee (y_i = y_j \wedge x_i < x_j)$$

Offensichtlich bleiben durch diese Erweiterung nicht nur alle Punkte entsprechend der gewählten Koordinate in der gleichen Reihenfolge, sondern wir haben auch eine *eindeutige* Reihenfolge. Darüber hinaus, kann keine Gleichheit zwischen verschiedenen Punkten gelten.

2D-Baum. Betrachten wir nun eine Datenstruktur, die unser anfangs eingeführtes Bereichssuche-Problem auch tatsächlich asymptotisch schneller zu lösen vermag, als der naive Ansatz.

Die Grundidee ist es, die Methoden die wir aus dem eindimensionalen Problem kennen für mehrere – zunächst zwei – Dimensionen zu erweitern. Bei der binären Suche betrachteten wir immer das Element „in der Mitte“ – den Median also –, und durchsuchten dann entweder nur die linke oder nur die rechte Hälfte weiter. Genau die selbe Methode kam auch bei den binären Suchbäumen zum Einsatz: Jeder Knoten enthielt einen Schlüssel; der linke Teilbaum enthielt alle Elemente mit kleineren Schlüsseln, der rechte Teilbaum enthielt alle mit größeren Schlüsseln.

Ebendiese Idee machen wir uns nun auch bei den sogenannten *2D-Bäumen* zunutze. Da wir aber nicht mehr nur ein Schlüsselkriterium besitzen, sondern zwei (die x - und die y -Koordinate) wenden wir diese stets abwechselnd an (vergl. Abbildung 8.4).

Zunächst suchen wir zwischen allen Punkt den Median bezüglich der x -Achse heraus, und machen diesen Punkt $u = (x_u, y_u)$ zur Wurzel unseres Baumes. Der Punkt u teilt nun die

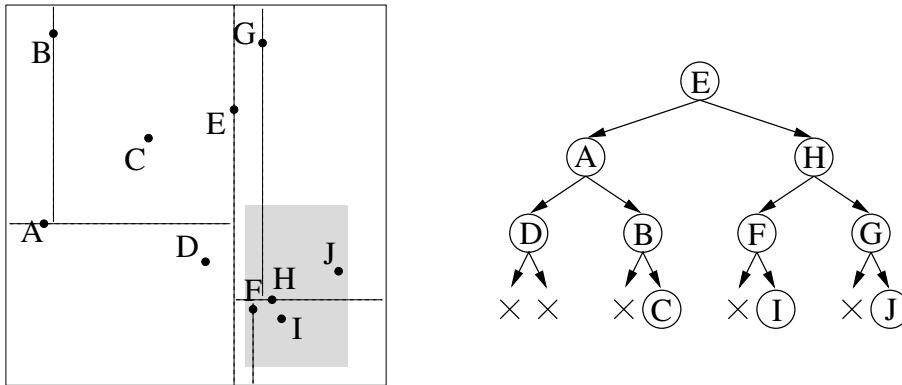


Abbildung 8.4: 2D-Baum.

Punktmenge in zwei fast gleichgroße Teilmengen auf, wobei die eine nur Punkte links von u (x -Koordinaten kleiner x_u), die andere nur Punkte rechts von u enthält.

Für jede Teilmenge überprüfen wir nun, ob sie noch mehr als ein Element hat; ist dem so, dann suchen wir in der Teilmenge wieder ein Medianelement v , diesmal aber in Bezug auf die y -Achse. Die Teilmenge wird also nun in zwei Unterteilmengen zerlegt, wobei die eine oberhalb, die andere unterhalb von v liegt. Enthält eine solche Unterteilmenge wieder noch mehr als ein Element, so wird sie abermals nach einem neuen x -Median aufgespalten, usw.

Es entsteht also ein binärer Baum, an dem an den Knoten mit ungerader Tiefe nach der y -Koordinate, bei denen mit gerader Tiefe (inklusive 0) nach der x -Koordinate aufgeteilt wird.

Bereichssuche. Basierend auf diesem Prinzip, ist der Bereichssuche-Algorithmus (Listing 8.2) nicht besonders kompliziert: Wir starten an der Wurzel und testen ob der zugehörige Punkt im Suchbereich D liegt. Danach überprüfen wir ob Teile von D links und/oder rechts von diesem Punkt liegen, und müssen dann entsprechend den linken und/oder rechten Teilbaum weiter durchsuchen. Dieses Suchverfahren setzt sich rekursiv fort, wobei entsprechend dem 2D-Baum-Prinzip immer zwischen x - und y -Koordinatenaufteilung hin- und hergewechselt wird.

Analyse der Bereichssuche. Da wir den Baum per Definitionem so aufgebaut haben, dass wir die Mengen immer jeweils halbiert haben, handelt es sich bei unserem 2D-Baum um einen perfekt balancierten binären Baum. Jeder Knoten im Baum entspricht genau einem Punkt aus P und umgekehrt. Daher hat der Baum n Knoten und eine logarithmische Tiefe. Ausgehend davon darf man hoffen, dass die Bereichssuche durch $O(\log n + R)$ abgeschätzt werden kann. Allerdings gelang es bisher noch nicht, diese Schranke formal zu beweisen; es existiert aber ein – an dieser Stelle leider zu umfangreicher – Beweis, der zeigt, dass die Suchzeit auf jeden Fall durch $O(\sqrt{n} + R)$ beschränkt ist. Schon dieser Term macht deutlich, dass die 2D-Baum-Methode deutlich effizienter ist, als der anfangs eingeführte naive Ansatz.

Eingabe: Suchbereich D und Wurzel \mathcal{B} des 2D-Baums
Ausgabe: Alle Punkte im 2D-Baum die in D liegen

```

1: procedure BEREICHSSUCHE(Wurzelknoten  $\mathcal{B}$ )
2:   HELPER_BEREICHSSUCHE( $\mathcal{B}$ , 'y')
3: end procedure

4: procedure HELPER_BEREICHSSUCHE(Knoten  $N$ , Koordinatenbezeichnung  $b$ )
5:   if  $N \neq \text{NULL}$  then
6:     if  $N.\text{punkt} \in D$  then gib  $N.\text{key}$  aus
7:     Ändere  $b$  von 'x' auf 'y' bzw. umgekehrt
8:     if  $D.\text{minPunkt} <_b N.\text{punkt}$  then HELPER_BEREICHSSUCHE( $N.\text{left}$ ,  $b$ )
9:     if  $D.\text{maxPunkt} >_b N.\text{punkt}$  then HELPER_BEREICHSSUCHE( $N.\text{right}$ ,  $b$ )
10:   end if
11: end procedure

```

Listing 8.2: 2D-Baum Bereichssuche

Aufbau eines 2D-Baums. So bleibt uns schließlich nur noch zu zeigen, wie wir einen 2D-Baum effizient aufbauen können; Listing 8.3 zeigt den notwendigen Pseudocode, der allerdings einiger Erklärung bedarf.

Die Schwierigkeit beim Aufbau-Algorithmus liegt in der effizienten Aufteilung der Mengen anhand der Mediane gemäß den alternierenden Dimensionen. Die Lösung des Problems besteht in zwei Arrays X und Y , die die Punkte aus P enthalten. Wir sortieren zunächst X nach den x -Koordinaten und Y nach den y -Koordinaten der Punkte, erhalten also nun zwei unterschiedliche Reihenfolgen der Punkte. Es ist klar, dass wir den Median gemäß der x -Achse einfach finden können, in dem wir das mittlere Element aus X auswählen; für die y -Achse können wir analog vorgehen.

Abbildung 8.5 verdeutlicht den im folgenden beschriebenen Vorgang. Sei p der Medianpunkt gemäß der x -Achse, so erzeugen wir einen Knoten im Baum der zu p korrespondiert, und dessen linker Teilbaum T_L alle Elemente enthält, die links, und dessen rechter Teilbaum T_R alle Elemente enthält die rechts von ihm liegen. Nun müssen wir die Punkte im Y -Array entsprechend *partitionieren*, sodass alle Punkte aus T_L in der linken, und alle Punkte von T_R in der rechten Hälfte des Y -Arrays landen. Innerhalb dieser Hälften muss allerdings die Sortierung nach der y -Koordinate erhalten bleiben. Den Punkt p setzen wir genau in die Mitte des Y -Arrays, da er schon behandelt ist. Eine derartige Partitionierung ist einfach in linearer Zeit zu bewerkstelligen.

Nun können wir rekursiv die beiden Teilbäume erzeugen; dazu benutzen sie jeweils nur „ihre“ Hälfte der Arrays. Wir stellen fest, dass für diese Teilbereiche alle Punkte nach x - bzw. y -Koordinate sortiert in X bzw. Y bereitstehen. Die Teilbäume suchen nun ihrerseits ihren Medianpunkt – allerdings aus den Teilarrays in Y – heraus, usw.

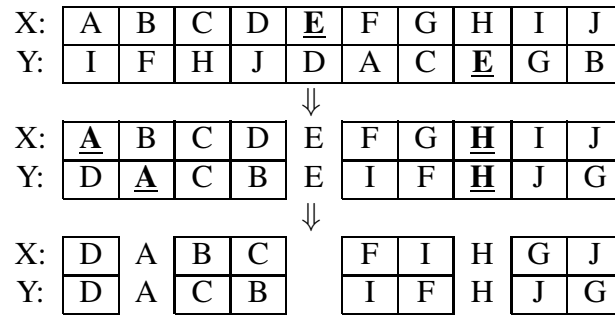


Abbildung 8.5: Beispiel für effiziente Medianaufteilung: Zunächst partitionieren wir Y gemäß dem Median der x -Koordinate (E) und erhalten die sortierten Teilmengen des linken und des rechten Teilbaums. Danach partitionieren wir X für deren Unter-
 teilbäume gemäß den Medianen der y -Koordinaten (A und H). Dies wiederholen wir
 so lange, bis die betrachteten Teilmengen maximal ein Element enthalten.

Analyse des Baumaufbaus. Zunächst stellen wir fest, dass das initiale Sortieren des X - und des Y -Arrays in $O(n \log n)$ möglich ist. Wie wir sehen werden, birgt die weitere Analyse gewisse Ähnlichkeiten mit der MergeSort-Analyse in sich: Wir wissen bereits, dass ein balancierter 2D-Baum, so wie wir ihn erzeugen, eine logarithmische Tiefe besitzt. Beim Erzeugen des Wurzelknotens partitionieren wir ein gesamtes Array der Länge n . Für die beiden Knoten auf Ebene 1 müssen wir jeweils ein halbes Arrays partitionieren, usw. Allgemein gibt es auf Ebene t immer $\Theta(2^t)$ Knoten, die jeweils $\Theta(\frac{n}{2^t})$ Elemente in $\Theta(\frac{n}{2^t})$ Zeit partitionieren. Daher benötigen wir in jeder Ebene $\Theta(n)$ Schritte. Dies bedeutet eine Laufzeit von $\Theta(n \log n)$ für die Summe über alle Ebenen – also für den gesamten Baumaufbau.

Dynamisch. Ähnlich wie bei einem binären Suchbaum besteht die Möglichkeit, Punkte – auf Kosten der Balance – dynamisch zur Datenstruktur hinzuzufügen. Auch das Entfernen von Punkten ist möglich, erfordert jedoch einen recht grossen Aufwand, weshalb man in der Realität diese Punkte meist im Baum belässt und zu jedem Punkt mitspeichert ob er aktiv oder inaktiv ist. Erst wenn sehr viele Punkt entfernt worden sind, bzw. die Balance durch Hinzufügen von Punkten zu stark gestört wurde, wird der Baum komplett neu aufgebaut. Es sollte jedem allerdings klar sein, dass dieser Ansatz mehr aus der Praxis heraus motiviert ist, als aus theoriebasierten Effizienzüberlegungen.

Höhere Dimensionen. Wir können den obigen Ansatz sehr einfach auf höhere Dimensionen anpassen, indem wir reihum nach den verschiedenen Dimensionsachsen aufteilen. D.h. bei $k = 3$ teilen wir z.B. an der Wurzel gemäß der x -Koordinate auf, auf Tiefe 1 gemäß der y -Koordinate, auf Tiefe 2 gemäß der z -Koordinate, auf Tiefe 3 wieder gemäß der x -Koordinate, usw. Die so entstehende Baumstruktur wird kD -Baum genannt. Man kann zeigen, dass der Aufbau eines solchen Baums eine Laufzeit von $\Theta(kn \log n)$ benötigt und die Bereichssuche

Eingabe: Punktmenge P , mit $|P| = n$

Ausgabe: Wurzel eines balancierten 2D-Baum

```

1: var X,Y : Feld von n Punkten

2: procedure AUFBAU(P)
3:   Befülle X und Y jeweils mit P
4:   Sortiere X nach x-Koordinaten
5:   Sortiere Y nach y-Koordinaten
6:   return HELPER_AUFBAU(1, n, 'x' )
7: end procedure

8: procedure HELPER_AUFBAU(Index l, Index r, Koordinatenbezeichnung b)
9:   var p : Punkt
10:  if l ≤ r then
11:    m = ⌈ $\frac{l+r}{2}$ ⌉
12:    if b = 'x' then
13:      p := X[m]
14:      PARTITIONIERE_FELD(Y, l, r, p)
15:      b := 'y'
16:    else
17:      p := Y[m]
18:      PARTITIONIERE_FELD(X, l, r, p)
19:      b := 'x'
20:    end if
21:    N := neuer Knoten mit N.punkt = p
22:    N.links := HELPER_AUFBAU(l, m - 1, b)
23:    N.rechts := HELPER_AUFBAU(m + 1, r, b)
24:    return N
25:  else
26:    return NULL
27:  end if
28: end procedure

```

Listing 8.3: Aufbau eines 2D-Baums

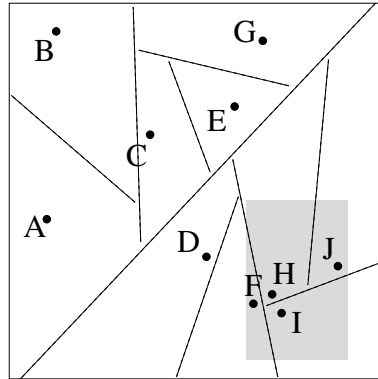


Abbildung 8.6: Aufteilung durch einen BSP-Baum.

$O(kn^{1-\frac{1}{k}} + R)$ Zeitschritte in Anspruch nimmt.

BSP-Baum. Die Idee der 2D-Bäume lässt sich zu einer Baumstruktur erweitern, die als *BSP-Baum (Binary Space Partition Tree)* bekannt ist. Die Idee ist, nicht immer regelmässig nach der x - bzw. y -Koordinate aufzuteilen, sondern an jedem inneren Knoten des Baums eine beliebige Gerade zu speichern, die die Fläche in zwei Halbfächen zerteilt. Im linken Teilbaum befinden sich dann alle Objekte die links von dieser Gerade liegen, im rechten Teilbaum befinden sich alle Objekte die rechts von dieser Gerade liegen. Die Objekte selbst werden als Blätter im Baum repräsentiert (vergl. Abbildung 8.6). Auch diese Datenstruktur findet viele Anwendungen in der Computergraphik, und lässt sich auch sehr einfach für höhere Dimensionen nutzen: im dreidimensionalen Raum z.B. definiert jeder innere Knoten eine Schnittebene, die den Raum in zwei Halbräume aufteilt, usw.

Weiterführende Literatur

Das Buch „Algorithms“ von Robert Sedgewick (siehe Literaturliste) bietet einen sehr gut geschriebenen Einstieg in die Welt der geometrischen Algorithmen, und beschäftigt sich unter anderem auch speziell mit den in diesem Kapitel vorgestellten Problemstellungen.

Wer noch mehr und Spezielleres über die Thematik erfahren möchte, dem sei z.B. das Buch „Computational Geometry“ von de Berg, van Kreveld, Overmars und Schwarzkopf ans Herz gelegt.